

Collecting fine-grained use traces in any application without modifying it

Blandine Ginon, Pierre-Antoine Champin, Stéphanie Jean-Daubias

▶ **To cite this version:**

Blandine Ginon, Pierre-Antoine Champin, Stéphanie Jean-Daubias. Collecting fine-grained use traces in any application without modifying it. workshop EXPPORT from the conference ICCBR, Jul 2013, New York, United States. hal-01339195

HAL Id: hal-01339195
https://hal.science/hal-01339195

Submitted on 7 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An approach for collecting fine-grained use traces in any application without modifying it

Blandine Ginon[1, 2], Pierre-Antoine Champin[1, 3]
and Stéphanie Jean-Daubias[1, 3]

[1] Université de Lyon, CNRS,
[2] INSA-Lyon, LIRIS, UMR5205, F-69621, France
[3] Université Lyon 1, LIRIS, UMR5205, F-69622, France
{name}.{surname}@liris.cnrs.fr

Abstract. We propose a technique to collect use traces in any existing application, without a need to modify this application. This technique is based on the use of accessibility libraries. We implemented our technique in a collector that uses UIAutomation and JavaAccessibility libraries: it can monitor Windows target-applications to collect the user's traces. The traces are then stored in a trace-base management system in order to be exploited thereafter. We have tested our collector on more than fifty applications in order to evaluate our approach.

Keywords: Use traces, event detection, accessibility.

1 Introduction

There is a growing interest in collecting traces of the interaction of a user with computer applications, for various purposes, like trace analysis [6], trace visualization [2] and trace-based assistance [20]. However, most applications are not designed to collect traces and it would be costly to redevelop them when a need to do so appears. Furthermore, the people that need to collect traces in an application do not always have its source code, if even they wished to modify it.

In the context of the AGATE project (Approach for Genericity in Assistance To complEx task) that aims at facilitating the use of complex software, without constraints on this software, we exploit the user's traces to provide personalized assistance. For this reason, we propose a technique to collect use traces in any application, without a need to modify it. This technique is based on the use of accessibility libraries that make possible the subscription to different kinds of events, like the mouse entered on an image or the selection of an item in a combo box, in order to know when those events occur, but also to know on which component of the user-interface they occurred. We implemented this technique in a collector that uses two accessibility libraries that target Windows applications: UIAutomation and JavaAccessibility.

After presenting related work in section 2, we present our approach in section 3, and we describe in section 4 how we implemented it. Section 5 discusses a prelimi-

nary evaluation of our implementation. Finally, we conclude and propose future improvements.

2 Related work

There is an abundant corpus of work on the analysis of logs and traces [5], [12-14], [21]. Historically, log analytics has first been dedicated to focus on the behavior of programs, for debugging or monitoring purposes. Then, the potential of using it to analyze the user's activity has been gradually recognized. Data mining and machine learning techniques have therefore been used for discovering processes in computer-mediated activities [3], [18], and more recently on identifying communities in social networks based on the user's interaction patterns [17].

Statistical and/or synthetic analysis is not the only way to exploit activity traces. Activity traces can also be considered as a repository of individual experiences that can be reused in a similar context, either identically or after an adaptation [4]. This is the underlying assumption of a number of efforts, such as those aiming at providing recommendations to users based on past experiences [8], [11], or monitoring the progression of a student in e-learning applications [16]. With the increasing availability of mobile devices and wearable sensors, practices of tracing various aspects of one's day-to-day life are also developing. Known as lifelogging [15] or quantified self [19], those practices aim at a better self-awareness or recollecting past events.

Depending on their intended tasks, the different approaches cited above require different kinds of events recorded in the respective traces. Except for lifelogging application (which are focused on real-world information acquired via sensors), most approaches rely on relatively high-level events (i.e. run application, open file). It follows that available tools for collecting interaction traces[1] are limited to capturing those high-level events. We believe, however, that some applications, such as personalized user assistance, require more fine-grained traces.

3 A technique to collect use traces

We propose a technique to collect fine-grained use traces in any existing application, that we will call a target-application. It has been stated in the previous section that this is already possible for high-level events. There are also tools to collect individual clicks or keystroke[2], but the only contextual information they provide is the application in which those events happened. By contrast, we want to be able to associate each traced event with a component of the user-interface of the target-application. For example, knowing on which button or menu item the user clicked is much more informative about his/her activity than only recording the application in which he/she clicked.

---

[1] For example http://dev.nepomuk.semanticdesktop.org/ or http://intersectalliance.com/snareagents

[2] For example https://github.com/gurgeh/selfspy or http://www.mykeylogger.com/

In this work, the traces we consider are sequences of records describing events (event type, time stamp, and other attributes)[3]. For this purpose, our technique is based on the use of accessibility libraries. Those libraries were initially created to allow accessibility tools (such as screen readers or braille terminals) to get information about the applications, in order to make them more accessible to disabled people. Using these libraries, it is possible to subscribe to different kinds of events, in order to know when those events occur, but also to know on which component of the user-interface they occurred. Indeed, accessibility libraries provide access to the full hierarchy of GUI components available to the user, as illustrated by Fig. 1: the root element represents the screen and its children represent all the open application windows: the calculator, Regards [7], Google Chrome and Paint in the example. What's more, we can see that the desktop (Program Manager) contains four elements: the trash icon, a pdf file, the NetBeans icon and the jar file Regards.jar.

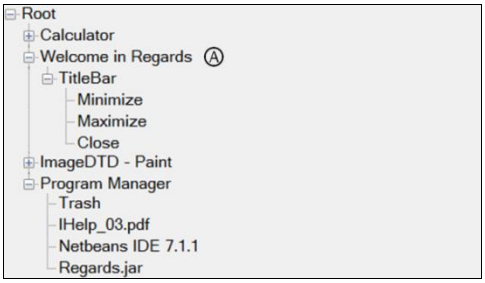


Fig. 1. Component tree detected by accessibility libraries

Thanks to accessibility libraries, we can also access information concerning each component of the user-interface: its type (window, button, check box…), its text, its position, its possible accessibility description… Note that, in the general case, a component has no persistent identifier, yet, when we record an event associated with a component of the user-interface, we need to be able to identify that component in the future uses of the trace. A solution is to characterize a component by its hierarchical position in the component tree and by additional information.

For this purpose, we consider the component hierarchy as an XML tree. Each XML element represents a component with its type, text, and accessibility description (if any). Each component is then characterized by an Xpath [1] containing all the relevant information about the component and its parents. As a simple example, Fig. 2 shows the XML tree describing the user interface of the Windows calculator (Fig. 3). Of all the available information about each component, we only keep their type, text and accessibillity description (if any). For example, we can see that not all buttons have a description, but that the button "CE" has one: "Clear entry" (cf. Ⓐ Fig. 2 and Ⓐ Fig. 3). This button can be characterized by the following Xpath:

```
//window[@type="CalcFrame" and @text="Calculator"]/
component[@type="Button" and @text="CE" and @description="Clear entry"]
```

[3] How these sequences are delimited (per session, per day, per application…) is out of scope, as our approach is neutral to that.

```
<Interface>
▼<window type="CalcFrame" text="Calculator">
  ▼<component type="TitleBar" text="Calculator">
     <component type="Item" text="Minimize"/>
     <component type="Item" text="Maximize"/>
     <component type="Item" text="Close"/>
   </component>
  ▼<component type="MenuBar" text="Application">
     <component type="Item" text="View"/>
     <component type="Item" text="Edit"/>
     <component type="Item" text="Help"/>
   </component>
   <component type="Button" text="MC" description="Memory clear"/>
   <component type="Button" text="MR" description="Memory restore"/>
   <component type="Button" text="MS" description="Memory store"/>
   <component type="Button" text="M+" description="Memory add"/>
   <component type="Button" text="M-" description="Memory removal"/>
   <component type="Static" text="12*3+"/>
   <component type="Static" text="4"/>
   <component type="Button" text="" description="Return"/>
   <component type="Button" text="C" description="Clear"/>
Ⓐ <component type="Button" text="CE" description="Clear entry"/>
   <component type="Button" text="," description="Decimal"/>
Ⓑ <component type="Button" text="" description="Negation"/>
   <component type="Button" text="/" description="Divide"/>
   <component type="Button" text="*" description="Multiply"/>
   <component type="Button" text="-" description="Substract"/>
   <component type="Button" text="+" description="Add"/>
   <component type="Button" text="" description="Square root"/>
   <component type="Button" text="" description="Percentage"/>
   <component type="Button" text="" description="Reciprocal"/>
   <component type="Button" text="=" description="Equal"/>
   <component type="Button" text="7"/>
   <component type="Button" text="4"/>
   <component type="Button" text="1"/>
   <component type="Button" text="0"/>
   <component type="Button" text="8"/>
   <component type="Button" text="5"/>
   <component type="Button" text="2"/>
   <component type="Button" text="9"/>
   <component type="Button" text="6"/>
   <component type="Button" text="3"/>
 </window>
</Interface>
```

Fig. 2. Description of the Windows calculator user interface.

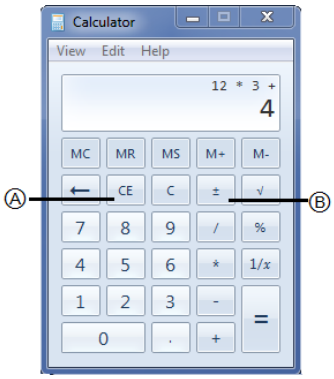


Fig. 3. Screenshot of the Windows calculator.

This description is of course redundant, but redundancy is important because each individual piece of information can be ambiguous in some contexts, as will be illustrated in Section 5. Note that the XML tree does not even need to be stored; it is just a way to formalize the information that is dynamically provided by accessibility libraries, and to justify the use of Xpath to address individual components.

Whenever an event is detected, it will be recorded in the trace with its type, the current time, the user's name and the Xpath characterizing the component on which the event appeared. Depending on the type of event, additional information can also be recorded (see Section 4). Let's come back to the example of the calculator. If the user enters the formula "12*3+4", the collector will detect a series of mouseClicked events on following buttons: "1", "2", "*", "3", "+" and "4". Other kinds of events may also be detected in the meantime (for example, mouseEntered events on the buttons hovered by the pointer during the moves between clicks).

4 Implementation of our technique

We implemented this technique in an operational collector that makes possible the collection of use trace in any existing Windows application, without a need to modify it. Our collector is based on two complementary accessibility libraries: the first one, UIAutomation [10], is aimed at Windows native applications, and the second one, JavaAccessibility [9], is aimed at Java applications.

4.1 Complementarity of accessibility libraries

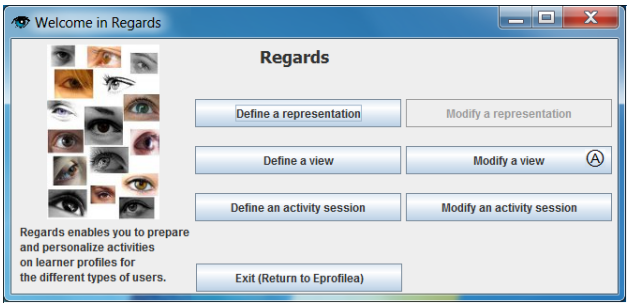


Fig. 4. Welcome screen of the Java application Regards.

UIAutomation detects any application running in Windows. However, for the case of Java applications, UIAutomation is only able to detect the frame of the application. Indeed, the inner components of Java applications are managed by the JVM (Java Virtual Machine) and not by Windows. As an example, we can see that only the frame with the title bar of the Java application Regards is detected (cf. Ⓐ Fig. 1). For this reason, we need a second collector for Java applications. We implemented it based on JavaAccessibleBridge and on the JavaAccessibility library, which is the equivalent of UIAutomation for the JVM. JavaAccessibility detects only applications running in the JVM, but contrarily to UIAutomation, it can detect their complete component tree. As an example, an extract of the interface description of the Java application named "Re-

gards" is given in Fig. 5. We can see that the welcome screen of Regards contains a button with a label "Modify a view" (cf. Ⓐ Fig. 5 and Ⓐ Fig. 4). The creator of Regards didn't associate any accessibility description to this button.

```
<Interface>
▼<window text="Welcome in Regards" type="class regard.Main" >
  ▼<component type="JRootPane" >
     <component type="JPanel" />
    ▼<component type="JLayeredPane">
      ▼<component type="JPanel" >
         <component text="Regards" type="JLabel"/>
         <component text="Define a representation" type="JButton"/>
         <component text="Modify a representation" type="JButton"/>
         <component text="Define a view" type="JButton" />
      Ⓐ<component text="Modify a view" type="JButton" />
         <component text="Modify an activity session" type="JButton" />
         <component text="Define an activity session" type="JButton" />
         <component text="Regards enables you to prepare and personalize activities
                          on learner profiles for the different types of users."
                    type="JLabel" />
         <component text="Exit (Return to Eprofiléa)" type="JButton"/>
       </component>
     </component>
   </component>
  </window>
  ...
</Interface>
```

Fig. 5. Extract of the interface description file for the Java application Regards.

4.2 Storing and managing traces

Fig. 6 shows the main events that our collector can detect using UIAutomation and JavaAccessibility. For instance, our collector can detect when the end user of the target-application clicks on a button (*mouseClicked*), when he/she moves the mouse pointer over an image (*mouseEntered*), when he/she selects an element in a combo box (*elementSelected*), and when he/she deselects a check box (*propertyChanged*). For some of these events, our collector detects complementary information. For instance, for the event *tooltipOpened*, our collector detects the text of the *tooltip*, and for the event *propertyChanged*, our collector detects the name of the property that changed (like *enabled*, *size*, *itemCount*, *rowCount*, *selected*, *visible*…) and the previous and new value of this property. Our collector can also detect additional information about a component depending on its type (Is the component enabled, selected, checked, collapsed, editable? Has it got the focus? What are its position and dimension, its value, its font and background color? ...).

The traces gathered by our collector are stored in a system called kTBS[4]. kTBS is an open-source implementation of a Trace-Based Management System (TBMS) [4] [20] developped in our team. It is a RESTful service, accessible through the HTTP protocol. Our collector sends the events to record to kTBS through an HTTP-POST request. The resulting traces can then be retrieved through an HTTP-GET request. This makes our collector relatively independent of kTBS; it can store traces in any other TBMS as long as they comply with the same protocol.

[4] http://liris.cnrs.fr/sbt-dev/ktbs/

It is worth mentioning that TBMS are not only meant to store traces: they are also able to compute transformed traces that provide different points of view on the traced activity, at different levels of abstraction. This is why our collector is only focused on low-level events; it relies on the TBMS to provide higher-level traces if needed.

Events		UIAutomation	JavaAccessibility
About **action**	Performed	✓	✓
About **mouse**	Clicked	✓	✓
	Entered	✓	✓
	Moved	✓	✓
	Pressed	✓	✓
	Released	✓	✓
	Dragged	✓	✓
	Exited	✓	✓
About **key**	Typed	✓	✓
	Released	✓	✓
	Pressed	✓	✓
About **focus**	Gained	✓	✓
	Lost	✓	✓
	Changed	✓	✓
About **menu**	Selected	✓	✓
	Deselected	✓	✓
	Opened	✗	✓
	Closed	✗	✓
About **tooltip**	Opened	✓	✗
	Closed	✓	✗
About **window**	Opened	✓	✓
	Closed	✓	✓
About **text**	Changed	✓	✗
	SelectionChanged	✓	✗
About **element**	Selected	✓	✓
	AddedToSelection	✓	✗
	RemovedFromSelection	✓	✗
About **property**	Changed	✓	✗

Fig. 6. Main events detected with UIAutomation and JavaAccessibility.

5 Evaluation and Discussion

In order to test our approach, we have successfully used our collector in more than fifty varied applications[5] (Windows native and Java, created by different developers). The aim of this test was to confirm if our collector can really collect events (see Fig. 6) in these applications; and to verify if the component concerned by each event can be identified in the component tree, with a view to make possible a rich exploitation of the collected traces. During this evaluation, we encountered problems in reliably identifying components. First, notice that if several components of the same type are at the same level in the hierarchy (a very frequent situation), only their text and de-

[5] List of those applications available at http://liris.cnrs.fr/blandine.ginon/detection.html

scription allow us to distinguish them. Note also that some components have no text associated to them, but only an image. In that case, only the accessibility description makes it possible to distinguish the component. This is consistent with the initial purpose of accessibility descriptions: to provide a text for accessibility tools (e.g. screen readers) when the component has no text of its own, or when the text is not descriptive enough.

In the example of the Windows calculator (cf. Fig. 2), several buttons have no label but only an icon, like the buttons "negation" and "square root". As a consequence, in our description file, we can distinguish these buttons only thanks to the description associated with the buttons. The Windows calculator has been developed with accessibility in mind, but unfortunately, this is not the case for all applications. Indeed, making an application accessible is time-consuming, and many developers prefer spending that time at adding new features to the application.

Accessibility descriptions themselves are not enough if they are not carefully chosen. Consider the case of Regards; the button Ⓐ Fig. 5 has no label, but only an icon representing an eraser and its description is "erase", like all the other buttons with an eraser in that window. As a consequence, it is not possible with our technique to distinguish those buttons from each other. Indeed, they will have exactly the same Xpath. This problem would not exist if the creator of the application had provided a more specific description for each button; "erase the view for the activity 'Visualize his profile' " for instance.

One way to overcome the lack of (good enough) accessibility descriptions is to use additional information to characterize components, especially their position in the window (which is also made available by accessibility libraries). We decided not to resort on that solution for the following reasons. First, the position of components within the window, and relatively to each other, may vary depending on a number of parameters (display settings, font size, window size). Second, setting good accessibility description is what developers should do anyway. So we prefer to encourage good practices, rather than compensate the absence of reliable information (accessibility description) by an alternative that may prove just as unreliable (position).

Another problem we encountered is that some applications are still not well detected by either the techniques of our collector. Those applications seem to manage their components without relying on the Microsoft foundation classes, and so are not correctly detected by UIAutomation. Most of them appear to be using the GTK toolkit[6], so a solution would be to add a part of our collector dedicated to GTK (as we did for Java).

Finally, another limitation of our approach is that it is based on localized information: texts and descriptions of components vary depending on the language of the interface. It is however technically quite simple to "translate" Xpaths in a language to another language, using a translation table as the one used internally by the application. For open-source software, this is even simpler as those translation tables are usually available in a standard format[7].

---

[6] http://www.gtk.org/

[7] http://www.gnu.org/software/gettext/manual/gettext.html#PO-Files

6 Conclusion and perspectives

There is an increasing number of applications that make possible interesting exploitations of use traces. For this reason, it is interesting to collect traces and to store them with a view to exploit them thereafter. We propose a technique to collect fine-grained use traces in existing applications, without a need to modify these applications.

The strengths of our technique are the accuracy of the use traces that it can collect, and its genericity. The more accurate the use traces, the more rich and varied will be the possible exploitation of these traces. Contrarily to existing techniques that collect clicks or keystrokes with very little contextual information, our technique associates each traced event with the concerned component from the user-interface of the target-application. What's more, our technique is not specific to an application but it can collect use traces in any application without a need to redevelop this application, even if they are not designed to collect use traces. On the other hand, as our technique is based on the use of accessibility libraries, it is impeded by the lack of accessibility features of some applications. Indeed, if a developer doesn't make any effort to make her application accessible, useful information can be missing, like the description of images.

In order to demonstrate the feasibility of our technique, we have implemented it in a collector that uses UIAutomation and JavaAccessibility. This collector can monitor Windows native applications and Java applications. We have tested this collector on more than fifty various applications from the simplest, like the Windows calculator, to the most complex, like the IDE NetBeans. These tests have showed the overall efficiency of our technique to collect fine-grained use traces in very varied applications.

We are currently working at the implementation of our technique with other accessibility libraries in order to make possible the collection of use traces in GTK Windows applications, as well as in Linux and Mac OS.

Acknowledgments. The authors would like to thank warmly Amélie Cordier for her participation to this work, in particular for her remarks and advices.

References

1. Berglund, A. *et al*. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation. http://www.w3.org/TR/xpath20/ . (2010).
2. Clauzel, D., Sehaba, K. and Prié, Y. Modelling and visualising traces for reflexivity in synchronous collaborative systems. In International Conference on Intelligent Networking and Collaborative Systems (INCoS 2009), Barcelona, Spain. pp. 16-23. IEEE Computer Society Los Alamitos, CA, USA. ISBN 978-0-7695-3858-7. (2009)
3. Cook, J. E., Wolf, A. L. :. Discovering Models of Software Processes from Event-Based Data. In: ACM Transactions on Software Engineering and Methodology, 7(3), 215–249. (1998)
4. Cordier, A., Lefevre, M., Champin, P-A., Georgeon, O., Mille, A. Trace-Based Reasoning: Modeling interaction traces for reasoning on experiences. In: 26th International FLAIRS Conference, St. Pete Beach, Florida, USA. (2013)

5. Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. Patterns in property specifications for finite-state verification. In: Software Engineering, 1999. Proceedings of the 1999 International Conference on (pp. 411-420). IEEE. (1999)
6. Fuchs, B. and Belin, A. Trace-Based Approach for Managing Users Experience. In Workshop TRUE: "Traces for Reusing Users' Experience". In: ICCBR 2012 TRUE and Story Cases Workshop, Luc Lamontagne, Juan A. Recio-Garc ed. Lyon, France. pp. 173-182. (2012)
7. Ginon, B., Jean-Daubias, S.: Models and tools to personalize activities on learners profiles. In: Ed-Media, Lisbon, Portugal (2011)
8. Groza,, T.; Handschuh, S., Möller, K., Grimnes, G., Sauermann, L., Minack, E., Mesnage, C., Jazayeri, M., Reif, G., Gudjonsdottir. R. The NEPOMUK Project -- On the way to the Social Semantic Desktop. In: Proceedings of the Third International Conference on Semantic Technologies (I-SEMANTICS 2007), Graz, Austria. (2007)
9. Harper, S., Khan, G., Stevens, R.: Design Checks for Java Accessibility. In: Accessible Design in the Digital World, Dundee, Scotland (2005)
10. Haverty, R.: New accessibility model for Microsoft Windows and cross platform development. In: ACM SIGACCESS Accessibility and Computing, pp 11-17. (2005)
11. Hug, C., Deneckere, R., Salinesi, C.: Map-TBS: Map process enactment traces and analysis, In: International Conference on Research Challenges in Information Science (RCIS), Valencia : Espagne (2012)
12. LeBlanc, T. J., Mellor-Crummey, J. M., & Fowler, R. J. Analyzing parallel program executions using multiple views. Journal of Parallel and Distributed Computing, 9(2), 203-217. (1990)
13. Lee, W., Stolfo, S. J., & Mok, K. W. A data mining framework for building intrusion detection models. In Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on (pp. 120-132). IEEE. (1999)
14. Mansouri-Samani, M., Sloman, M.: GEM: a generalized event monitoring language for distributed systems. In: Distruted Systel Engineering, 4(2). (1997)
15. O'Hara, K., Tuffield, M. M., & Shadbolt, N. Lifelogging: Privacy and empowerment with memories for life. Identity in the Information Society, 1(1), 155-172. (2008)
16. Poltrack, J., Hruska, N., Johnson, A., & Haag, J. The Next Generation of SCORM: Innovation for the Global Force. In The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC) (Vol. 2012, No. 1). National Training Systems Association. (2012)
17. Sachan, M., Contractor, D., Faruquie, T. a., Subramaniam, L. V.: Using content and interactions for discovering communities in social networks. In: Proceedings of the 21st international conference on World Wide Web - WWW '12, pp 331-341. (2012)
18. Song, M., Günther, C. W., Van der Aalst, W.: Trace Clustering in Process Mining. In M. Van der Aalst, M. *et al.* (Eds.), Business Process Management Workshop (pp. 109–120). Springer Berlin Heidelberg. (2009)
19. Wolf, G., Carmichael, A., & Kelly, K. The quantified self. TED http://www.ted.com/talks/gary_wolf_the_quantified_self. html . (2010)
20. Zarka, R., Champin, P-A. , Cordier, A., Egyed-Zsigmond, E. , Lamontagne, L., Mille., A. TStore: A Web-Based System for Managing, Transforming and Reusing Traces. In: ICCBR 2012 TRUE and Story Cases Workshop, Luc Lamontagne, Juan A. Recio-Garc ed. Lyon, France. pp. 173-182. (2012)
21. Zhang, Y., and Wenke L. Intrusion detection in wireless ad-hoc networks. In Proceedings of the 6th annual international conference on Mobile computing and networking, pp. 275-283. ACM. (2000)